

AD-A249 192



RL-TR-91-274, Vol II (of five)
Final Technical Report
November 1991



PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Larch/Ada Rationale

ORA Corporation



Sponsored by
Strategic Defense Initiative Office

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Strategic Defense Initiative Office or the U.S. Government.

92-11200



Rome Laboratory
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

02 4 27 487

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

Although this report references limited documents listed below, no limited information has been extracted:

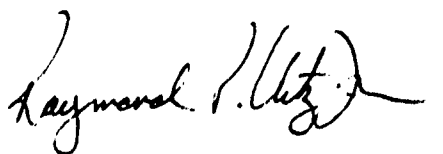
RL-TR-91-274, Vol IIIa, IIIb, IVa, and IVb, November 1991. Distribution authorized to USGO agencies and their contractors; critical technology; Nov 91.

RL-TR-91-274, Vol II (of five) has been reviewed and is approved for publication.

APPROVED:


JOHN C. FAUST
Project Engineer

FOR THE COMMANDER:


RAYMOND P. URTZ, JR.

Director
Command, Control and Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL(C3AB) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

PENELOPE: AN ADA VERIFICATION ENVIRONMENT,
Larch/Ada Rationale

David Guaspari
Ian Sutherland

Contractor: ORA Corporation
Contract Number: F30602-86-C-0071
Effective Date of Contract: 19 Aug 86
Contract Expiration Date: 30 Sep 89
Short Title of Work: PENELOPE: AN ADA VERI-
FICATION ENVIRONMENT, Larch/Ada Rationale
Period of Work Covered: Aug 86 - Aug 89
Principal Investigator: Maureen Stillman
Phone: (607) 277-2020
RL Project Engineer: John C. Faust
Phone: (315) 330-3241

Approved for public release; distribution unlimited.

This research was supported by the Strategic Defense Initiative Office of the Department of Defense and was monitored by John C. Faust, RL/C3AB, Griffiss AFB 13441-5700, under contract F30602-86-C-0071.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE November 1991		3. REPORT TYPE AND DATES COVERED Final Aug 86 - Aug 89	
4. TITLE AND SUBTITLE PENELOPE: AN ADA VERIFICATION ENVIRONMENT, Larch/Ada Rationale				5. FUNDING NUMBERS C - F30602-86-C-0071 PE - 35167G/63223C PR - 1070/B413 TA - 01/03 WU - 02	
6. AUTHOR(S) David Guaspari and Ian Sutherland					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) ORA Corporation 301A Dates Drive Ithaca NY 14850-1313				8. PERFORMING ORGANIZATION REPORT NUMBER ORA TR 17-8	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Strategic Defense Initiative Office, Office of the Secretary of Defense Wash DC 20301-7100 Rome Laboratory (C3AB) Griffiss AFB NY 13441-5700				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-91-274, Vol II (of five)	
11. SUPPLEMENTARY NOTES RL Project Engineer: John C. Faust/C3AB/(315) 330-3241					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This rationale provides a careful, but informal, account of the meanings of Larch/Ada specifications. It also describes the simplifying assumptions under which proofs in the Penelope verification environment imply that an Ada program satisfies its Larch/Ada specification. The account is intended to be adequate to the needs of a Penelope or Larch/Ada user, and to serve as an introduction for mathematically motivated readers who wish to consult other documents detailing the mathematics of Penelope.					
14. SUBJECT TERMS Ada, Larch, Larch/Ada, Formal Methods, Formal Specification, Program Verification, Predicate Transformers, Ada Verification				15. NUMBER OF PAGES 48	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL		

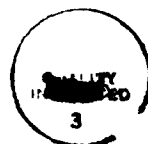
Contents

1	Introduction	1
2	Two-Tiered Specifications	2
2.1	Verification	2
2.2	Assertion Language	3
2.3	An Example of Two-Tiered Specification	4
2.3.1	The mathematical component	5
2.3.2	The interface component	5
2.4	Abstraction and Reuse	6
3	Larch/Ada	7
3.1	The Larch Shared Language	8
3.2	The Larch/Ada Interface Component	9
3.3	Relation to Anna	10
4	Larch/Ada Terms and Ada Objects	11
4.1	Integer Types	12
4.1.1	The Values of Type <code>integer</code>	13
4.1.2	Larch/Ada Terms and their Evaluation	16
4.1.3	Integer objects	18

4.2	Modeling Ada types	21
4.2.1	Values	21
4.2.2	Larch/Ada terms	22
4.2.3	Objects	23
4.2.4	Implementing Larch/Ada: theories of Ada types	23
5	Mathematical Foundations of Larch/Ada	24
5.1	Foundations for Penelope	24
5.1.1	Ada and Ada'	24
5.1.2	Program errors	25
5.2	Formalizing the semantics of Larch/Ada	26
5.2.1	The meaning of the mathematical component	26
5.2.2	Specifications and proofs	26
5.2.3	Proof obligations	27
6	Appendix: Simplifying Assumptions	28
7	Appendix: Models of Ada Types	29
7.1	Logic	30
7.2	Discrete types	30
7.2.1	Operations	30
7.2.2	Explanations	31

7.3	Enumeration types	32
7.4	Integer types	33
7.5	Array types	33

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



1 Introduction

Penelope is a system for formally verifying Ada programs. The formal specification of an Ada program is expressed in Penelope in a language called *Larch/Ada*. This document explains the semantics of Larch/Ada. The goals of this explanation are:

- to provide guidance in interpreting Larch/Ada specifications so that someone reading a specification can better determine exactly what the specification says about the program
- to record the reasons for certain decisions which were made in the course of designing Larch/Ada (many of which were based on semantical considerations)

The major Larch/Ada design decision was to use the Larch approach to program specification, which is described in [7] and [11]. This approach separates a specification language into two “tiers”, one of which is used to describe purely mathematical objects relevant to specification and the other of which is used to specify the behavior of programs using these mathematical objects. Section 2 gives a general description of the two-tiered approach. Sections 3 and 4 describe how the two-tiered approach is applied in Larch/Ada. Section 5 gives an introduction to the mathematical foundations of Larch/Ada. A more detailed formal exposition can be found in [6].

Sections 6 and 7 are appendices. Section 6 describes some of the simplifying assumptions which apply to verifications in Penelope. Section 7 describes how various Ada types are modeled in Penelope using the two-tiered approach.

David Luckham’s work with Anna [9] inspired the first version of Larch/Ada, which was accordingly called Polyanna. We have retained many of his ideas. The name has changed because our specification language has become, in the terminology of Guttag, Horning, and Wing, a “Larch interface language.” We have borrowed liberally from their work in [7] and [10].

The reader of this manual should know some Ada and should be familiar with formal verification of computer programs, at least at the level of the books by David Gries [4] or Edsger Dijkstra [3]. That requires, in particular, modest familiarity with ordinary first-order many-sorted logic. An acquaintance with Anna [9], with the Larch approach to program specification [11], and with Hoare's paper [8] on proving the correctness of data representations, would also be helpful.

Throughout this manual a "specification" is a prescription of program behavior. This usage does not conform to the practice of the Ada Reference Manual, which uses the word "specification" for what might, in more generic terminology, be called the "header" or "signature" of a subprogram or package. When we wish to refer to such headers we will explicitly call them "*Ada specifications*."

2 Two-Tiered Specifications

2.1 Verification

Penelope is based on the Floyd/Hoare or verification condition (VC) method for verifying programs. In this method, a program is specified by *entry and exit conditions*. An entry condition is an assertion about the state in which the program is called. An exit condition is an assertion about the state in which the program terminates. The program is *partially correct* if all possible executions of the program satisfy the following requirement: *if* the program is invoked in a state in which the entry condition is true, *and* the program terminates (normally or exceptionally), *then* the exit condition is true in the state in which it terminates.

To verify a program using the VC method, the user first supplies a collection of *annotations*. An annotation is an assertion about the state of the program at some point of control internal to the program. In particular, the user supplies annotations called *loop invariants* which state that a certain assertion

about the program's state is true every time control reaches a certain point inside a loop.

From an annotated program—that is, a program together with its specification and user-supplied annotations—Penelope generates a set of formulas of first-order logic, called the *verification conditions* (VCs) of that annotated program. Under certain assumptions (described in Section 6), the program is partially correct if all of the VCs are true. To verify the program, the user proves the VCs using Penelope's proof editor.

Penelope allows the user to associate exit conditions with various different kinds of termination, such as normal termination and termination with user-defined exceptions. Penelope also allows the user to associate annotations with various different points internal to a program. These are described in [2]. In this document we will be primarily concerned with the language for making assertions about program states (the *assertion language*).

2.2 Assertion Language

Penelope's assertion language is based on first-order logic. Basing an assertion language on first-order logic presents us with an immediate problem however. First-order logic languages were created to make statements about mathematical objects and structures, but we want to use them to make statements about computational objects. Computational objects behave differently from mathematical objects in several ways. For example, if a first-order language contains a symbol for a function from integers to integers, the semantics of first-order logic requires that the function be total, that is, defined on all integers. An Ada function which takes an integer and returns an integer may not be total because it may not terminate on all possible inputs. For example, integer addition on Ada will not be a total function, since, on any machine, there will be pairs of integers which, when added, cause an overflow.

One possible solution to this problem is to extend first-order logic so that it can talk about computational objects. For example, we can introduce

a special “undefined object” to model the possibility of non-termination, and “error objects” to model the raising of exceptions. This was our initial approach to specifications in Penelope, but it was abandoned when we found that the logic for talking about computational objects would have to be very complex. In particular, reasoning about the resulting verification conditions tends to be quite difficult.

Instead, we adopt the Larch “two-tiered” approach to program specification. The two-tiered approach separates each specification into two parts: a *mathematical component* and an *interface component*. The mathematical component describes a mathematical structure, and the interface component uses this structure to express entry and exit conditions and annotations. The link between the mathematical component and the interface component is that the mathematical component describes operations and predicates on certain sets (such as the set of integers), and program objects, such as program variables, take their values from these same sets.

2.3 An Example of Two-Tiered Specification

We will illustrate the two-tiered approach by an example. Consider a function

```
function factorial(x : integer) returns integer;
```

intended to compute the mathematical factorial function on non-negative inputs. We will give a semi-formal specification of this function in the form of a mathematical component and an interface component.

2.3.1 The mathematical component

Let *fact* be any totally defined mathematical function on the (infinite set of) mathematical integers that satisfies the axiom:

$$fact(x) = \begin{cases} 1 & \text{if } x = 0 \\ x * fact(x - 1) & \text{if } x > 0 \end{cases}$$

The mathematical component of our specification consists of the above axiom on *fact*, together with all the usual arithmetical operations on the mathematical integers.

Note that our axiom says nothing about the values of *fact* on negative integers. Since *fact* is a total, mathematical function on the integers, *fact*(*x*) must be some integer when *x* < 0, but our mathematical component does not specify anything about such values.

2.3.2 The interface component

The following entry-exit condition, expressed in English, serves as the interface component of our specification of factorial:

This function may not be invoked unless $x \geq 0$. If $0 \leq x$ and the function returns a value, then it returns *fact*(*x*) as its value.

In an actual Larch/Ada specification, of course, English phrases like "This function may not be invoked unless" and "If ... then it returns" would be replaced by keywords of Larch/Ada. For example, the requirement that the function only be invoked on nonnegative integers would be expressed by

IN $x \geq 0$

which is Larch/Ada's syntax for specifying entry conditions.

Now let's consider how this specification illustrates the two-tiered approach.

The specification makes a formal separation between mathematical objects and computational objects. The function *fact* is a mathematical entity, while the denotation of *factorial* (in whatever programming language semantics we might write down for our programming language) is a computational object. The two are not the same; in fact, they belong to different worlds.

The connection between the two components arises from the fact that the values that are passed to and returned by *factorial* are modeled by mathematical integers. The computational object represented by *factorial* is not in the world of mathematical functions, but the value of the parameter *x* which is passed to *factorial* and the value which is returned by *factorial* (if it returns a value) are. It is therefore perfectly well-defined to talk about whether the value of *x* passed to *factorial* is ≥ 0 or not, because this value is a mathematical integer. It is also well-defined to talk about whether the value returned (if any) is equal to *fact(x)*, because both of these are mathematical integers.

We are able to specify *factorial* even though the mathematical component does not provide an "undefined" object to represent the possibility that it fails to terminate, and does not provide "error" objects to represent the possibility that it may terminate by raising an exception. The interface component of the specification uses the mathematical component to specify what must happen when the function *does* terminate normally, and says nothing about the other cases.

2.4 Abstraction and Reuse

An additional feature of two-tiered specification is that it supports use and re-use of abstractions. For example, there is a useful "ideal" or "mathematical" notion of stack (unbounded, last-in-first-out) in terms of which it is easy to describe many different actual stack implementations—implementations which vary in size and in the behavior they exhibit under anomalous circumstances, such as attempts to push onto a full stack or to pop an empty

one. Two-tiered specification provides a systematic way in which to isolate such widely applicable idealizations, and apply them to the description of particular cases.

Consider the example of `factorial` above. The same mathematical tier can easily be used to specify a variety of other programs, such as:

- a function that raises the exception `negative` when the input is negative
- a function that raises the exception `too_big` when the input is positive and the program does not terminate normally
- a function that returns as its value $fact(x) + 1$
- a procedure that has a side effect on a global variable `u`, replacing the value of `u` by $fact(u)$

Anyone who understands the meaning of the mathematical component immediately understands the meanings of all these other specifications. No new conceptual work is necessary. Any collection of lemmas, rewrite rules, etc., generated from the definitions of the mathematical component is available for the proofs of any these programs.

3 Larch/Ada

In this Section we give some of the specifics of how the two-tiered approach is applied in Larch/Ada.

Larch/Ada consists of constructs for both specifying the externally observable, input-output behavior of a program, and for writing internal embedded assertions and loop invariants. Adopting Anna's terminology, we call all such constructs *annotations*. The word *specification* will continue to refer to external specifications—implementation-independent prescriptions of program

behavior. The semantics of the assertion language is the same for both external specifications and internal embedded assertions.

3.1 The Larch Shared Language

The mathematical component of a Larch/Ada specification is a theory in standard many-sorted first-order logic. That is, the user introduces specification concepts like *fact* by providing a first-order theory that axiomatizes their properties. The notation in which that theory is described is the Larch Shared Language ([7]).

Such theories will not be intelligible unless they are presented in highly organized ways—indicating, for example, how “small” constituents are combined into larger composite theories, or indicating the desired logical relations between theories.

The Larch Shared Language provides a notation, the *trait* construct, for presenting first-order theories in ways that make their logical and conceptual organization explicit. The denotation of a trait Tr is a first-order many-sorted theory Th .

The user is permitted, but not encouraged, to write Tr in a completely *unstructured* way: as a list of function signatures, followed by a list of axioms about those functions. However, the Larch Shared Language permits the user to construct Tr out of subexpressions indicating how its denotation Th is assembled from other theories, and to make assertions about the relations between them. For example, subexpressions of Tr may say any of the following things:

- The axioms of the theory Th include all the axioms of Th' (where Th' is some other theory denoted by a trait).
- Th introduces no new assumptions about the basic arithmetical operations.

- Any formula of Th containing occurrences of the symbol f can be rewritten as a formula that does not contain f .

The Larch Shared Language is therefore a tool for helping the specifier to construct a mathematical tier that properly expresses his intentions, and is intelligible to other users. For details we refer the reader to [7].

The Larch Shared Language is called “shared” because it is intended to be the mathematical component for many different two-tiered specification languages (for different programming language for example). The difference between these specification languages will be their interface components, which provide facilities for stating properties of programs in terms of the traits of the mathematical component.

3.2 The Larch/Ada Interface Component

The interface component of a Larch/Ada specification has the following syntactical form: Certain keywords, corresponding to phrases like “may not be invoked unless,” are followed by appropriate *terms*. Terms may denote values (as in “*fact(x)*”) or express constraints on program states (as in “ $x \geq 0$ ”). Syntactically, terms are made up from the symbols introduced in the mathematical tier, logical operators (including quantifiers), and identifiers (including program variables and formal parameters) denoting Ada objects. We discuss terms further in Section 4.

Terms have mathematical, not computational, meaning. That is why we do not allow the identifier `factorial` to occur in terms, and why the logic of terms is simple. Dynamic, computational behavior, such as the raising of an exception, is indicated in an interface component by the appropriate keyword, and the “logic” of such behavior is built into Penelope’s VC generator.

3.3 Relation to Anna

Both syntactically and conceptually Larch/Ada is a great deal like Anna [9]. Anna is a formal discipline for inserting comments (*annotations*) into Ada programs. All the annotations of Larch/Ada have analogs in Anna. The difference between analogous Anna and Larch/Ada constructs is semantic: the Anna constructs analogous to Larch/Ada terms have a computational, rather than mathematical, meaning.

A legal Ada program with comments satisfying the rules of Anna's syntax is called an Anna program. The Ada program is called the Anna program's *underlying Ada text*. An Anna program can be transformed into an Ada program that runs the underlying text and, in passing, checks each state to see that it satisfies the constraints expressed in its annotations. The transformed program raises `anna_error` if it detects the occurrence of a state violating these constraints. So long as execution of the transformed program does not raise this exception, its effects (aside from a loss of efficiency) should be identical with those of the underlying Ada text.

Anna can therefore be thought of as an extension of Ada with extra checking constructs, which compile into Ada, and whose semantics is defined in terms of the execution semantics of Ada.

We abandoned our initial plan to formalize the logic of Anna because the constructs of Anna are so thoroughly computational—making its underlying logic rather complex. Indeed, if one takes the execution of the translated Anna program as the definition of Anna's semantics, then there is no way to formalize the logic of Anna apart from a formalization of the semantics of Ada.

We remain indebted to Anna for much of our notation and terminology and refer the reader to the Anna Reference Manual [9] for more information.

4 Larch/Ada Terms and Ada Objects

This Section describes the syntax and semantics of Larch/Ada *terms*, and the related notions of *assertions* and *states*. Syntactically, a term is a well-formed sequence of symbols in our particular version of first-order logic. Semantically, terms have mathematical, not computational, meaning. They are distinct from Ada expressions, which are part of the imperative Ada language. Terms denote values. In particular, every possible value of every Ada object is denoted by some term.

In general, the value of a term depends on the state in which it is evaluated, although the values of some terms (like “ $1 + 1$ ”) are independent of states. The value of a term is never undefined or equal to some “undefined element.” This makes it possible to reason about terms in ordinary first-order logic.

An assertion is a term whose value is a boolean (*true* or *false*). An assertion can be regarded as expressing a constraint on states. The constraint is met if the value of the assertion is *true* in a given state, and is not met otherwise.

Via the “mathematical parts” of specifications, a user may introduce new symbols and thereby expand the set of terms more or less at will. This section deals only with the predefined vocabulary provided by Larch/Ada for denoting the values of Ada objects and describing the basic operations on them.

We will now describe the predefined terms of Larch/Ada. We will first illustrate the basic ideas for the case of type *integer*: how the values of integer types and subtypes are modeled, how integer objects are modeled, and how integer terms are evaluated. We will then describe the general approach for all Ada types.

4.1 Integer Types

The language of the mathematical tier contains sort symbols. A sort symbol denotes a set of values, which is called the *carrier* of the sort symbol. The carriers of the sort symbols of the mathematical tier are used to model the values of Ada objects. In the case of the type `integer`, the mathematical tier contains a sort symbol *Int*, whose carrier is the set of all the mathematical integers. The values of all variables or formal parameters of type `integer` are modeled as mathematical integers—i.e., as elements of the carrier of sort *Int*. In Larch terminology, this is expressed by saying that the type `integer` is *based on* the sort *Int*. Every Ada type mark is based on some (unique) sort symbol.

We associate with `integer` not only the sort symbol *Int*, but also several other sort symbols, and a collection of symbols for various functions. The collection of all symbols (sort and function) associated with the type `integer` will be denoted by Σ_{int} , and is referred to as a *signature*. We will describe Σ_{int} further below.

Also associated with `integer` is a class of *algebras* for the signature Σ_{int} . We will denote this class by \mathcal{A}_{int} . An algebra is an assignment of carriers to sort symbols and functions to the function symbols. The reason we must associate a *class* of algebras with type `integer` is that some of the predefined functions are not completely specified. This is discussed further below. Also, as seen in the `factorial` example, the user may introduce new function symbols which are not fully specified. Any time a function symbol is not fully specified, there will be a number of possible functions which can serve as its interpretation. Each of these possibilities gives rise to a different algebra. We will give an example of this below.

In each of the algebras in \mathcal{A}_{int} , the sort symbol *Int* has the mathematical integers as its carrier. We model the values of type `integer` as a subset of the values of the carrier of *Int* in \mathcal{A}_{int} .

4.1.1 The Values of Type integer

Here is a brief, and partial, description of \mathcal{A}_{int} .

Sorts The sorts of \mathcal{A}_{int} are *Int*, *Bool*, *AdaBool*, *AdaChar*, and *AdaString*. The carrier of *Int* in \mathcal{A}_{int} is the set of mathematical integers, and the carrier of *Bool* is the set of boolean values. The sort *AdaBool* is the sort on which the Ada type boolean is based. The technical reasons for distinguishing *Bool* and *AdaBool* are discussed in section 7. The sorts *AdaChar* and *AdaString* are those on which the Ada types *character* and *string* are based, respectively. Discussion of their carriers is deferred to section 7.

The sorts *AdaBool*, *AdaChar*, *AdaString*, and the operations on them are included because they are needed to describe the basic Ada operations of type *integer*. In fact, rather than modeling *integer* in isolation, we must model package *standard* as a whole. For present purposes we can usually ignore the non-integer sorts and operations.

Operations Here we list a few of the symbols in Σ_{int} and discuss their meanings (i.e., their interpretations in \mathcal{A}_{int}). So far as possible the operations are given mnemonic names, to indicate the role they play in defining Ada types and operations. A full listing of Σ_{int} is provided in section 7.

Arithmetical operations \mathcal{A}_{int} has a full complement of standard arithmetical operations (including the decimal numerals), with their usual mean-

ings. For example:

$$\begin{aligned} \# + \# &: \text{Int}, \text{Int} \rightarrow \text{Int} \\ \# - \# &: \text{Int}, \text{Int} \rightarrow \text{Int} \\ \# * \# &: \text{Int}, \text{Int} \rightarrow \text{Int} \\ \# / \# &: \text{Int}, \text{Int} \rightarrow \text{Int} \\ &\text{etc.} \\ 0 &: \rightarrow \text{Int} \\ 1 &: \rightarrow \text{Int} \\ 2 &: \rightarrow \text{Int} \\ &\text{etc.} \end{aligned}$$

The division operation is not normally provided in first-order formulations of arithmetic, because of the awkwardness of dealing with terms like "1/0." The way to think about terms like 1/0, 0/0, $x/0$, etc., in Larch/Ada is that they denote well-defined, but totally unspecified, integers. Accordingly, $1/0 = 1/0$ is true, because every integer is equal to itself. On the other hand, we may *not* assume that 1/0 equals 2/0 or that 0/0 equals 1. This is the example alluded to above of a predefined operation which is not fully specified. The values of $n/0$ for various n are not specified. Each possible choice of values for these terms gives rise to a different algebra in \mathcal{A}_{int} . In practice, we rarely have to remember that we are dealing with a class of algebras rather than a single algebra.

The fact that 1/0 has a value does not contradict the fact that Ada's division operation raises `numeric_error` when it attempts to evaluate 1/0. Since we insist that the values of Larch/Ada terms always be defined, we will be faced with many instances, like 1/0, of "nonsensical" terms. As mentioned above, these terms are taken care of consistently by the two-tiered approach. The Larch/Ada term 1/0 is a mathematical integer whose value we don't specify. It is completely separate from the Ada expression 1/0, which is not assigned a mathematical value because it is a computational object. Instead, Penelope contains rules about what happens when such an expression is evaluated.

Scalar operations The algebras in \mathcal{A}_{int} also “inherit” a number of operations applicable to the values of any discrete type, for example:

$\text{pred} : \text{Int} \rightarrow \text{Int}$
 $\text{succ} : \text{Int} \rightarrow \text{Int}$
 $\# <_b \# : \text{Int}, \text{Int} \rightarrow \text{Bool}$
 $\# <_a \# : \text{Int}, \text{Int} \rightarrow \text{AdaBool}$
 $\# =_b \# : \text{Int}, \text{Int} \rightarrow \text{Bool}$
 $\# =_a \# : \text{Int}, \text{Int} \rightarrow \text{AdaBool}$
etc.

The value of $\text{succ}(x)$ is $x + 1$, for every integer x , even though evaluation of the analogous Ada operation `integer'succ` may raise an exception.

Because we distinguish the boolean sort *Bool* from the sort, *AdaBool*, on which the type `boolean` is based, it is necessary to include two versions of relational operators like `<` and `=`. We use `<b`, for example, to make assertions like “ x is less than y ” and `<a` to denote the value returned by an Ada expression like “ $x < y$.”

Values in Other Integer Types and Subtypes In Larch/Ada, all integer types and subtypes are based on sort *Int* and therefore all take their values in $\mathcal{A}_{\text{int}}(\text{Int})$. It is clearly “right” that a type and its subtypes should take values in the same domain. In Penelope we go further, and base *all* the implementation-defined integer types on *Int*, and base any parent type and its derived types on the same sort. Here are some consequences of that decision.

After the declarations

```
type S is range 1..10;  
type T is range 5..20;  
x : S := 3;  
y : T := 4;
```

an Ada expression like $x = y$ is illegal. On the other hand, the Larch/Ada terms $x =_a y$ and $x =_b y$ are well-formed because, from the point of view of Larch/Ada, x and y take their values in the same sort. In Ada, one can compare x and y by means of explicit type conversion: $x = S(y)$ is legal. In Larch/Ada this type conversion is the identity function, so that evaluation of $x = S(y)$ corresponds, approximately, to evaluation of the Larch/Ada term $x =_a y$.

4.1.2 Larch/Ada Terms and their Evaluation

The notion of a Larch/Ada term is scoped, in the sense that each point in an Ada text is associated with a set of legal Larch/Ada terms, and that set varies from point to point. For the purposes of this discussion, we will make two simplifications: we ignore the scoping, and we restrict attention to the Ada texts whose only constants, variables, and formal parameters are of type *integer*.

Larch/Ada Terms Here is a first approximation to the definition of "Larch/Ada term": Treat each Ada *identifier* that is a program variable, formal parameter, or program constant of type *integer* as a logical constant of sort *Int*, and then apply the usual syntactic constructions of first-order logic to these logical constants and the symbols in Σ_{int} .

For example, let x and y be program variables of type *integer*. Then,

- Some Larch/Ada terms of sort *Int*:

$$0 \quad (5 * 4) / 3 \quad x \quad y \quad x + 6 \quad pred(x)$$

- Some Larch/Ada terms of sort *AdaBool*:

$$x <_a y \quad x =_a 3$$

- Some Larch/Ada terms of sort *Bool*:

$$x <_b y \quad \forall z : Int(z = z) \quad \forall z : Int(z + y = y + z)$$

(The definition given above is a simplification. We will give the complete definition in sections 4.1.2 and 4.1.3).

Simple program variables and constants A *simple variable*, or simple program variable, is an Ada program variable that is an identifier. Similarly, programming language constants that are identifiers are called *simple constants*.

It is important to note that the only Ada names that may occur in Larch/Ada terms are formal parameters, and those program variables or constants that are identifiers. For example, after

```
type A is array(integer) of integer;  
x : integer := 0;  
A : array := <others => 1>;
```

$A(x)$ is the name of a program variable, but is *not* a Larch/Ada term, and may not occur in one. Larch/Ada represents the value of such a variable by means of a complex term: the result of applying a (mathematical) selection operation to the two Larch/Ada terms A and x . Such terms are discussed further in section 7.

Evaluation of terms The value of a Larch/Ada term depends on the state in which it is evaluated.

States A state is a function that associates certain Larch/Ada identifiers with values. Among these identifiers are the simple program variables and constants, and they are our present concern. A state always assigns an actual value (not “undefined” or “error”) to every such identifier. The value associated with an uninitialized variable is discussed in section 4.1.3. Simple integer variables and simple integer constants are mapped to integers—i.e., to elements of $\mathcal{A}_{\text{int}}(\text{Int})$.

The other information represented in states, which will be discussed later, includes: historical information (e.g., whether a variable has been initialized), the values of implicit Ada objects (e.g., the heap associated with an access type), and user-defined state information stored in "virtual objects."

Evaluations The value of a Larch/Ada term t in state s is calculated by evaluating t in \mathcal{A}_{int} , with the simple program variables and constants have the values given to them by s . For example, if x has value 3 in state s , then in state s the Larch/Ada term $\text{succ}(x)$ has value $\mathcal{A}_{\text{int}}(\text{succ})(3)$ —that is, the value 4. Because a state assigns a value to every simple program variable, every Larch/Ada term also receives a value—and is never "undefined".

A Larch/Ada term may also refer to the initial value of a formal parameter. For example, if x is a formal in parameter of a subprogram then $\text{in } x$ is a Larch/Ada term denoting the value of x when the subprogram was invoked. Terms of this form are evaluated as follows: the evaluation of the term $\text{in } x$ in a state s is the evaluation of the term x in the state s_0 , where s_0 is the initial state of the subprogram or function in question. This is discussed further below.

4.1.3 Integer objects

Strictly speaking, Larch/Ada terms denote not Ada objects, but the values that such objects may contain. One of the ways in which an object differs from a value is that an object has a history. Therefore, in addition to terms associated with the values of particular objects, Larch/Ada contains terms associated with their histories.

A *simple object* is an object named by a simple variable or constant.

Formal parameters Formal parameters are not the names of true objects, but are treated as objects in certain contexts. They will be so treated in this discussion.

Histories of Simple Objects Larch/Ada contains two built-in ways to refer to the histories of simple integer objects.

Initialization According to Ada semantics, the result of the declaration

`x : integer;`

is that the object `x` exists but does not “have a value.” An immediate attempt to execute `x := x;` or to evaluate `x = x` would be an *erroneous* attempt to read a variable that does not “have a value.”

Larch/Ada represents these facts about the object `x` by pair of values:

- an integer, denoted by the Larch/Ada term $(x : \rightarrow Int)$;
If the variable has not been assigned to, we treat this value as not only uninteresting, but unobservable (in the sense that an attempt to read it is a catastrophic error).
- a boolean, denoted by the Larch/Ada term $(x'defined : \rightarrow Bool)$.
This assertion (i.e., Boolean term) expresses exactly what the Ada reference manual means by the phrase “`x` has a value.”

In the state immediately after the declaration, $(x : \rightarrow Int)$ is a “defined but unspecified” integer value and $(x'defined : \rightarrow Int)$ has value *false*. In any state after `x` has become initialized, $(x'defined : \rightarrow Int)$ evaluates to *true*.

Notice that immediately after the declaration of `x`, the assertion `x =b x` is satisfied and the value of `x =a x` is that element of *AdaBool* corresponding to the Ada object *true*. This is true despite the fact that an attempt to evaluate the executable expression `x = x` in that state is a program error.

A Larch/Ada term `x'defined` is also associated with each formal integer parameter `x`.

“In” values To specify the effects of subprogram executions one must be able to “remember” the values which the subprogram parameters and globals have on entry. So, for example, if x is a simple integer variable global to a procedure P , the fact that P has the side effect of incrementing x by 1 can be specified as

$$\text{out } (x = (\text{in } x) + 1)$$

The keyword “out” says that the assertion that follows is an assertion about the exit state of P . The leftmost occurrence of x denotes the value of x in the exit state, and $\text{in } x$ refers to the value of x on entry to P .

If x is a simple integer variable, then in certain contexts $\text{in } x$ and $(\text{in } x)'\text{defined}$ are Larch/Ada terms of sort *Int*. It is necessary to make $(\text{in } x)$ available for the formulation of subprogram specifications and convenient to make it available for formulating assertions about details of a subprogram’s execution.

The full logical declarations of these Larch/Ada terms are:

$$\begin{aligned} &(\text{in } x : \rightarrow \text{Int}) \\ &((\text{in } x)'\text{defined} : \rightarrow \text{Int}) \end{aligned}$$

Virtual Variables A virtual variable (sometimes called a “history variable”) does not denote a true object. It is a way for the user to direct Penelope to “remember” values that were calculated in previous states during the execution of a program. The term $x'\text{defined}$ can be regarded as a virtual variable declared and manipulated automatically, out of the user’s control. Notice that $x'\text{defined}$ has a sort, but not a type. This is characteristic of virtual variables.

Specification constructs representing “declarations of” and “assignments to” virtual variables can be introduced into Ada texts in places where ordinary Ada declarations and assignments might occur. For example,

```

--: x : Int := 0;      -- the '--:' is a syntactic flag
                        -- Int is a sort symbol, not a typemark
                        -- 0 is a Larch/Ada term

      while b loop
--: x := x+1;          -- x+1 is a Larch/Ada term
      ...
      end loop;

```

makes $(x : \rightarrow Int)$ into a counter that, upon exit, has recorded the number of iterations of the loop.

The semantics of virtual variables is analogous to, but not identical with, that of Ada variables. Virtual declarations and assignments do not represent executions, but can be modeled as modifications to the “virtual” part of the state.

One notable difference between the values of actual and virtual variables is that the value in every (actual) integer variable is constrained to lie between `minint` and `maxint`, whereas the value in a virtual integer value is unconstrained.

4.2 Modeling Ada types

A general account of Larch/Ada’s treatment of arbitrary Ada types and objects closely follows the account just given of the integer types and objects.

4.2.1 Values

Every Ada type T is associated with a corresponding sort S_T in the Larch/Ada term language, the sort on which it is based, and is also associated with a class of algebras \mathcal{A}_T . Objects of type T take their values in a subset—usually a proper subset—of $\mathcal{A}_T(S_T)$, the carrier of sort S_T in \mathcal{A}_T .

We denote the signature of \mathcal{A}_T by Σ_T . The operations of Σ_T are available to help describe the results of the basic operations on type T .

A type, its subtypes, and all its derived types are based on the same sort. In general, any two types that are explicitly or implicitly convertible are based on the same sort, and type conversion between them is the identity function.¹

4.2.2 Larch/Ada terms

In this discussion of Larch/Ada terms we fix an Ada program P and once again simplify things by ignoring scopes. Obtain the signature Σ by putting together all the signatures Σ_T for all the Ada types T occurring in P . The Larch/Ada terms for P are obtained, essentially, by applying the usual rules of first-order logic to construct terms and formulas from Σ and the following collection of symbols, all of them treated as logical constants of appropriate sorts:

- Simple program variables, constants, and formal parameters, and their corresponding "in" versions. Those of type T have sort S_T .
- x 'defined and (in x)'defined, of sort *Bool*, whenever x is a simple variable of a discrete type.
- Virtual variables, with the sorts provided by their declarations.

Note: One may declare a virtual variable of any sort, and not only of those sorts associated with type marks.

Every possible value of every Ada object in program P can be denoted by a Larch/Ada term that contains only symbols from Σ .

¹The exception to this is type conversion of array objects, since a conversion may involve altering the bounds of an array's index types.

States A state is a function assigning a value to every program variable and virtual variable. Each of these terms is associated with a sort, and the value associated with it must lie in the carrier of that sort.

When we speak of the *virtual part* of the state we mean the values associated with the user-supplied virtual variables. The rest of the state is the actual part. The effect of an Ada execution depends only on the actual part of the state (and does not alter the virtual part at all).

Evaluation of terms The value of an arbitrary Larch/Ada term in a given state is calculated just as in the case of integer. The state supplies the values of the variables, etc., and the algebras \mathcal{A}_T supply the meanings of all other symbols occurring in Larch/Ada terms. As a result, every term has a value in every state.

4.2.3 Objects

The apparatus of “in” variables and user-defined virtual variables is the same for integer as it is for all other types.

4.2.4 Implementing Larch/Ada: theories of Ada types

The official meaning of Larch/Ada terms like $x+y$ or $\forall z : \text{Int}(z+x = x+z)$ is the meaning they receive in the algebra \mathcal{A}_{int} . One ordinarily reasons about such terms from some axiomatic list of properties of \mathcal{A}_{int} . The report [5] explicitly shows how to formulate, for each Ada type T , a “useful” axiomatic theory Th_T that is satisfied by the algebra \mathcal{A}_T . These axioms are implemented in the Penelope system.

For every T , \mathcal{A}_T contains a representation of mathematical integers and all their basic operations, and therefore no axiomatic description of \mathcal{A}_T captures all its properties. The user should be aware that reasoning about Larch/Ada can always legitimately assume *any* property of Larch/Ada terms that is

true in \mathcal{A}_T , whether or not it is provable in first-order logic from the axioms supplied in Penelope. This means, in particular, that any other theorem prover that is sound for \mathcal{A}_T could be used in instead of, or in conjunction with, the Penelope prover.

5 Mathematical Foundations of Larch/Ada

This section is a very brief introduction to some features of the mathematics underlying Larch/Ada and Penelope: an outline, plus citations of papers containing the details.

5.1 Foundations for Penelope

A formal specification for the Penelope system would be something like: Declare a program verified if certain criteria are satisfied. A foundation for Penelope is an argument purporting to show that this specification is correct, in the sense that a program that satisfies those criteria really does behave as advertised.

5.1.1 Ada and Ada'

Our first problem is that Ada lacks a formal definition.² Accordingly, we formally define the semantics of a closely related language, Ada', *having the same syntax* as Ada. This definition uses standard techniques of denotational semantics. Though sequential, Ada' is non-deterministic (as is sequential Ada).

The semantics of Ada' is more tractable than that of Ada. For example, Ada' stipulates the methods of parameter passing even when Ada leaves them

²Neither the AdaEd interpreter nor the EEC-sponsored formal definition of Ada has official standing.

undetermined. We impose restrictions on program texts to help compensate for this difference by disallowing programs that are sensitive to the choice of parameter mechanism.

The argument that, in the end, we really have compensated for Ada's simplifications is necessarily informal. Experts will immediately wonder about the complex interactions between Ada's underdetermined parameter-passing mechanisms and exception-raising.

5.1.2 Program errors

Erroneous programs and incorrect order dependences deserve separate mention. These program errors are of two kinds.

The errors belonging to one class are detectable by consulting current state information. For example, any attempt to read an uninitialized variable is an error of this kind. The Ada semantics of such an execution is totally undefined, and therefore any Larch/Ada specification implicitly asserts that such errors do not occur. The VC's generated by Penelope require the user to prove that. We may view the Ada' semantics of these executions as the raising of a predefined, unhandleable, catastrophic exception.

The errors belonging to the other class are defined in terms of the *effect* of a program, where the notion of "effect" is left undefined. For example, a procedure call whose "effect" is sensitive to the order in which its parameters are evaluated is an error of this kind. The Ada semantics of such executions is various—in some cases the result is undefined. Our solution is to regard two different executions as having the same "effect" if they satisfy the same specifications. In other words, something is an effect only if it affects whether an execution meets the constraints of the specification. Given this definition, Penelope verifies that no program errors of either class occur.

5.2 Formalizing the semantics of Larch/Ada

The paper [6] provides, in a somewhat general setting, a formal definition of *satisfaction* of a two-tiered specification. This section contains some general remarks on the meaning of two-tiered specifications and their use in program proofs.

5.2.1 The meaning of the mathematical component

Return to the example of factorial. Notice that the mathematical component does not uniquely pin down the meaning of *fact*, since its value on negative inputs is completely undetermined.

The meaning of the two-tiered specification, however, is unambiguous. It says that the code must obey its interface specification no matter what interpretation of *fact* is chosen (so long as it obeys the axioms). Intuitively, this says that the meaning of the mathematical component is nothing but the consequences of its axioms.

5.2.2 Specifications and proofs

The formal definition of satisfaction is purely semantic. It makes no reference to provability, let alone to any particular proof system. Nonetheless, some intuition can be gained by understanding certain proof obligations that are *sufficient* to imply satisfaction.

For example, from what axioms does the proof of a VC take place? Suppose that program P_1 is implemented in terms of program P_2 , and that the mathematical component of the specification of each P_i is the theory Th_i . Essentially, the VC generated for P_1 is a formula whose symbols may come from either Th_1 or Th_2 ; and its proof takes place in the union of theories Th_1 and Th_2 .

Proving the VC for P_1 in $Th_1 \cup Th_2$ is not quite sufficient to show that P_1 satisfies its specification: If $Th_1 \cup Th_2$ is an inconsistent theory, such a proof would be vacuous. Th_1 and Th_2 must, therefore, be consistent with one another. There exist other constraints, in addition to this consistency requirement, on the way in which the mathematical components of specifications combine. These other constraints can be thought of as additional VC's for ensuring that multiple programs verified from multiple collections of traits are compatible.

5.2.3 Proof obligations

The use of a two-tiered specification assumes the consistency of its mathematical component. Other, subtler, assumptions are made when two-tiered specifications are combined—when, for example, the specifications to some program units are used as hypotheses to the proof of another.

Here is an illustration of one such requirement. Suppose that we write a program P in terms of some predefined type T . Plainly, the specification of P cannot legitimately introduce new assumptions about the behavior of basic operations on T . A proof of P from such unwarranted assumptions would be fallacious. This can be rephrased as follows: Suppose that the predefined operations of T are characterized by a two-tiered specification whose mathematical component is given by theory Th . It is illegitimate for the mathematical component of P to imply any formula from the language of Th unless that formula is also a consequence of Th .

There is no universally applicable technique for discharging obligations of this kind, and the current Penelope system does not generate these additional VC's.

6 Appendix: Simplifying Assumptions

Formal verification makes explicit what we mean by saying that a program is correct, makes explicit the hypotheses on which our belief in its correctness depends, and (presumably) strengthens that belief by eliminating or simplifying hypotheses we would otherwise have to make.

This section lists the restrictions and assumptions that qualify any verification, in Penelope, that an Ada program satisfies its Larch/Ada specification.

Penelope is applicable only to a subset of the legal Ada programs, those satisfying the following restrictions:

- no use of real number types or operations, unchecked programming, machine dependencies, tasking, or generics
- restrictions on subprogram calls to prevent improper aliasing of parameters against each other or against global variables
- restrictions on expressions to prevent improper use of side effect

One class of assumptions is important, but quite generic: A verification conducted on the source code assumes the correctness of the compiler's translation and of the all the levels (assembler, . . . , hardware) beneath it. However plausible or implausible it may be, this assumption is inescapable (and is also made, implicitly, by the ordinary programmer). The user of Penelope also assumes, of course, that his verification has not slipped through because of some bug in Penelope's code.

Our non-generic assumptions are as follows. A Penelope verification of an Ada program says nothing about an execution during which any of the following happens:

- Numeric or storage overflow occurs.

- A compiler-introduced optimization has altered the effect of the execution (other than by altering its efficiency).

The output of a program can be altered as a result of legal optimizations (See the example in [1], §11.6, paragraph 10.)

- The predefined exception `program_error` is raised in circumstances where its raising is merely optional.

The first of these requirements is straightforward: our mathematical model assumes infinite storage capacity. It means precisely what it says: An occurrence of `storage_error`, for example, invalidates a Penelope verification even if that occurrence is handled and therefore not propagated.

The restriction on optimizations is rather serious, since it would be extremely difficult to persuade oneself that a compiler satisfied it, unless that compiler satisfied some very strong restriction, such as: execution of the statements of the Ada program occurs in the canonical order.

7 Appendix: Models of Ada Types

The purpose of this section is to give the reader an intuitive picture of the values we associate with each Ada type and of the meanings of the predefined Larch/Ada operations on those values—that is, to describe \mathcal{A}_T for each type T .

The various \mathcal{A}_T 's have many parts in common—e.g., all contain the mathematical integers and booleans (and other things as well). It is therefore inconvenient, and redundant, to describe all of \mathcal{A}_T for each type T . Instead, we proceed by describing the values in S_T , and explaining the meanings of the Larch/Ada operations wherever it is most convenient to do so.

7.1 Logic

All the usual logical operations, such as boolean connectives and quantifiers, are available.

7.2 Discrete types

Let T be a discrete type. Our fundamental design decision is that the carrier of S_T is an *infinite* set which is isomorphic to the mathematical integers. In particular, this decision requires us to distinguish the two-element boolean sort *Bool* from the sort *AdaBool*, on which the discrete type *boolean* is based. The reason for this decision will be explained after more of the model is described.

7.2.1 Operations

The models of all discrete types have available the following Larch/Ada operations:

$<$ is a total ordering of S_T isomorphic to the ordering of the mathematical integers. The operations *pos* and *val*, which are made available in order to describe $T'POS$ and $T'VAL$, are isomorphisms between the order of S_T and the ordering of integers in *Int*. They are inverses of one another.

The *pred* and *succ* operations return the predecessor and successor, respectively, in the infinite ordering $<$.

The operations *image* and *value* are used to describe the operations $T'IMAGE$ and $T'VALUE$, respectively.

7.2.2 Explanations

Although it seems reasonable to base `integer` on *Int*, it seems odd to base an enumeration type with two elements on an infinite set isomorphic to the integers. We do so because it is simpler. We provide one example of the complications that arise if we attempt to base some or all of the discrete types on sorts whose carriers are finite: the interaction between discrete types and array types.

Example: Slides Many array operations (including array assignment) involve sliding an array—rigidly translating it—along its indices. For example, the following fragment is legal

```
declare
  type I is (r,o,y,g,b,i,v);
  type T is array(I range <>) of integer;
  A : T(o..g);
  B : T(b..v);
begin
  ...
  A := B;
end;
```

As a result of this assignment, `A(o)` becomes equal to `B(b)`, `A(y)` to `B(i)`, and `A(g)` to `B(v)`. One way to say this is to say that `A` becomes equal to the array “`B` slid 3 places downward.” The definition of sliding is simple when the index type is based on an ordered set that is infinite both directions, and complicated when the index set is based on a finite ordering. Complications arises when one attempts to slide an array “off the end” of a finite index set. In this example, if the carrier of `I` contains only seven elements, one would run off the end of the index set by trying to slide `B` 5 places downward.

Conclusion There are many other cases in which it proves simpler to base the discrete types on infinite orderings—and these simplifications do not seem to be mere trade-offs displacing the complications elsewhere. The reason is that the complications introduced by using finite carriers are typically introduced for the sake of cases that never arise in practice. We never need to compute the result of sliding an array off the end of its index type. One way or another, a request to slide an array is always preceded by some kind of test to see whether it “fits”—and if not, the slide isn’t carried out.

The sole drawback is the need to distinguish *Bool* from *AdaBool*.

7.3 Enumeration types

After the declaration

```
type T is (red,blue,green);
```

Σ_T makes available names for the elements of T

```
(red :  $\rightarrow S_T$ )
(blue :  $\rightarrow S_T$ )
(green :  $\rightarrow S_T$ )
```

and Th_T guarantees that *succ*, *pred*, *pos*, etc. behave as expected on these three elements:

```
succ(red) = blue
pred(green) = blue
pos(blue) = 1
etc.
```

The axioms of Th_T also guarantee that the *value* and *image* operations behave like T’VALUE and T’IMAGE when the Ada functions return without raising exceptions.

7.4 Integer types

In addition to the operations possessed by all sort symbols for discrete types, Σ_{int} includes the standard integer arithmetic operations with their usual meanings. The values of computations like $x/0$, $2 * (-3)$, $\text{mod}(x, 0)$, and $\text{rem}(x, 0)$ are left unspecified.

7.5 Array types

For simplicity's sake we discuss only one-dimensional arrays (which are all that Penelope currently supports).

Consider as our first example an array of discrete elements:

```
type I is range 1..20;  
type T is array(I range <>) of integer;  
A : T(5..10);
```

$(A : \rightarrow S_T)$ is a Larch/Ada term. A potential value for A —i.e., a value in the carrier of S_T —contains information of various kinds:

- It associates *every* value in the carrier of the index type with a value in the component type. The value that A associates with i is denoted by $A[i]$. $A[7]$ is the value contained in the Ada object $A(7)$, if it has a value, and $A[11]$ is an integer value of no observational significance.
- Its index bounds are constituents of its value.
- It associates *every* value in the carrier of the index type with a value of sort *Bool*. The boolean $A@7$ indicates whether the Ada object $A(7)$ has a value. (Booleans like $A@11$ are of no observational significance.)

We can think of the values of S_T as being generated by the following two operations:

- The value of *makearray*(*i*, *j*) is a completely uninitialized array whose lower index bound is *i* and whose upper index bound is *j*. (It is not required that $i \leq j$.) To say that it is totally uninitialized is to say that for every integer *k*, *makearray*(*i*, *j*)@*k* = *false*. Immediately after its declaration, *A* has value *makearray*(1,5).
- The value of *A*[*i* \Leftarrow *j*] differs from that of *A* in at most two ways: its *i*th component is initialized (whereas that of *A* may or may not be initialized), and the value of its *i*th component is *j*.

The basic picture for arrays whose components are not discrete elements differs only in that we effectively omit the "@" operation by leaving it totally unspecified.

Our definition of the set of values in S_T is insensitive to whether *T* is a constrained or an unconstrained array type. This is consistent with previous design decisions, since every constrained array type is a subtype of an unconstrained array type (which may be anonymous).

References

- [1] ANSI. *The Programming Language Ada Reference Manual*, 1983. ANSI/MIL-STD-1815A.
- [2] Odyssey Research Associates. *The Larch/Ada reference manual*. Technical Report TR-17-8, Odyssey Research Associates, 1989.
- [3] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, 1976.
- [4] David Gries. *The Science of Programming*. Springer-Verlag, 1981.
- [5] David Guaspari. *Domains for Ada types*. Technical Report TR-17-11, Odyssey Research Associates, 1989.

- [6] David Guaspari. Semantics of two-tiered specifications, part I: modular programming. Technical Report TR-17-12, Odyssey Research Associates, 1989.
- [7] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report TR 5, DEC/SRC, July 1985.
- [8] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(1):271-281, 1972.
- [9] D. C. Luckham et al. Anna: A language for annotating Ada programs. Technical Report CSL-84-261, Stanford University, 1986. Reference Manual.
- [10] Jeannette M. Wing. *A Two-tiered Approach to Specifying Programs*. PhD thesis, MIT, Cambridge, Massachusetts, 1983.
- [11] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1-24, January 1987.

**MISSION
OF
ROME LABORATORY**

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C³I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C³I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.